A Convenient Framework for Efficient Parallel Multipass Algorithms

Markus Weimer, Sriram Rao, Martin Zinkevich
Yahoo! Labs
Santa Clara, CA 95054
[weimer|sriramr|maz]@yahoo-inc.com

1 Introduction

The amount of data available is ever-increasing. At the same time, the available time to learn from the available data is decreasing in many applications, especially on the web. These two trends together with limited improvements in per-cpu speed and hard disk bandwidth lead to the need for parallel machine learning algorithms. Numerous have been proposed in the past (including [1, 3, 4]). Many of them make use of frameworks like MapReduce [2], as it facilitates easy parallelization and provides fault tolerance and data local computation at the framework level. However, MapReduce also introduces some inherent inefficiencies when compared to message passing systems like MPI.

In this paper, we present a computational framework based on Workers and Aggregators for data-parallel computations that retains the simplicity of MapReduce, while offering a significant speedup for a large class of algorithms. We report experiments based on several implementations of Stochastic Gradient Descent (SGD): The well known sequential variant as well as a parallel version inspired by our recent work in [5] which we implemented both in MapReduce and the proposed framework.

2 Example: Multi-Pass Parallel SGD

For the purposes of this algorithm we consider the set of examples as a set of convex functions $\{c_1, \dots c_m\}$. The algorithm we provide runs as follows:

Algorithm 1 $\mathrm{MultiPassParallelSGD}(\{c^1, \dots c^m\}, T, \eta, w^*, k)$

Divide the data randomly among k machines with $m' = m/k$ examples on each machine.
$\forall$ machines $j, \forall$ examples t, $c_t^j \leftarrow$ the tth example sent to machine j.
Initialize a common weight vector w^*.
for all $i \in \{1 \dots T\}$ **iterations do**
  for each machine $j \in \{1, \dots k\}$ **parallel do**
    Push $w^j \leftarrow w$.
    Choose a permutation $p : \{1 \dots m'\} \to \{1 \dots m'\}$ uniformly at random. (shuffle the data)
    for all $t \in \{1 \dots m'\}$ **do**
      $w^j \leftarrow w^j - \eta \nabla c^j_{p(t)}(w^j)$
    end for
  end for
  $w^* \leftarrow \frac{1}{k} \sum_{j=1}^{k} w^j$
end for

A key distinction between this and [5] is that after the first pass, data is not shared in a random fashion. Thus, the theory from that paper does not directly follow. However, as we lower η, the difference between the initial and final w^* in a pass drops. Thus, for small η, the algorithm approximates batch gradient descent.

Algorithm 2 Worker/Aggregator

INIT: Partition Data D; All workers i receive a handle to a partition D_i
for all $j \in \{1, \dots T\}$ **iterations do**
  Each worker performs $v'_i = f_{D_i}(v)$ and returns v'
  Each aggregator performs $v = g(\{v'_1, \dots, v'_i\})$ and returns v
end for
return the last v

3 Architecture

Algorithm 1 can easily be implemented in a MapReduce framework: Each of the map instances represents one of the inner SGD passes: After loading w_{j-1}, each of the machines running the map function (the mappers) runs SGD over the data assigned to it. After its pass it emits the new weight vector which is subsequently averaged by the reduce function running on a single or multiple machines. This implementation enjoys all the benefits of MapReduce implementations, such as:

Fault Tolerance: The MapReduce framework monitors mappers and reducers and restart tasks on other machines should one of the machines fail.
Data Locality: Typically, MapReduce is applied on top of a distributed filesystem. Thus, the framework can schedule the mappers on machines "close" to the data in the network topology.

This implementation of the algorithm is, however, sub-optimal from a performance perspective: The algorithm repeats a procedure for T iterations between which only the relatively small model w changes. This structure is far from unique to the parallel SGD studied in this paper. Many algorithms perform data-parallel passes interleaved with coordination steps: batch gradient descent, bundle methods and conjugate gradient descent obviously share this structure. Conjugate gradient descent and bundle methods, computationally speaking, differ most radically from other methods in that they use a line search, requiring several global decisions and therefore a lot of communication. However, there is a trick where you take the projection of each function along the direction of search, communicate these compressed examples to a central server, and then do the line search.

The MapReduce implementation cannot make use of this knowledge: (a) In each iteration, the framework needs to schedule and provision the map and reduce machines. (b) In each iteration, all the data is read again from disk, the cost of which can easily dominate the cost of the relatively simple SGD updates. (c) As different iterations are treated as independent jobs, they need to communicate by storing and reading w from disk, which also adds strain to the distributed file system.

To address these issues, we propose a different architecture composed of *Workers* and *Aggregators*:[1]

Worker: Upon initialization, a worker receives a handle to its data D, which it may load into memory. The worker is then applied to a status v upon which it returns a status v'.
Aggregator: An Aggregator receives all v' from the workers and aggregates them to a new status v, which is then distributed to the Worker machines.

Workers and Aggregator communicate directly through the network. The whole process is shown in Algorithm 2. Of course, such a setup is trivially achievable within a message passing system. However, the formulation of the algorithm in terms of Workers and Aggregators gives rise to similar benefits as in the MapReduce framework. Besides greatly simplifying the formulation of parallel algorithms in comparison to using MPI, this approach also enjoys data locality and fault tolerance: *Data locality* can be achieved by scheduling the Workers "close" to their data. By monitoring the input and output of the workers and aggregators, the framework can both detect and recover from machine failures easily.[2] Besides retaining the major benefits of MapReduce, this model avoids the drawbacks of it outlined above: In the absence of failures, no machines are rescheduled between iterations. And given enough memory in all the workers combined, even the data loading is not repeated between iterations, yielding additional speedups.

[1] We omit the trivial extension to a Combiner-equivalent here for space considerations.

[2] For many stochastic algorithms such as the one proposed here and under the assumption that a Worker failure is unrelated to the data processed by that machine, a recovery may simply be to spin up another machine with that data for the next round.

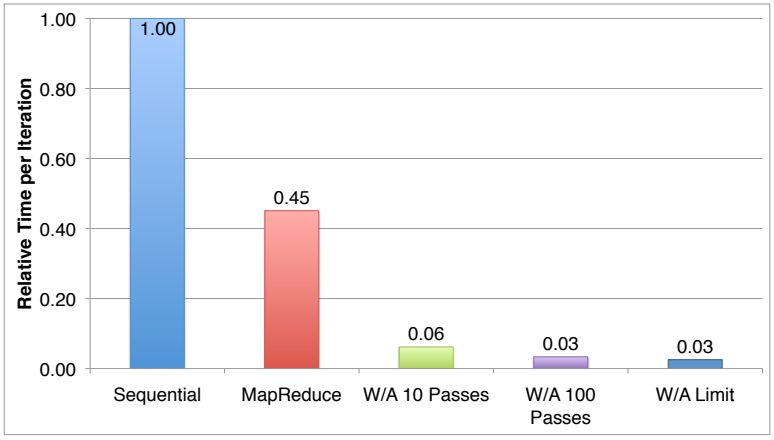

Figure 1: Relative training time per pass of the different systems (smaller is better): Sequential, MapReduce, Worker/Aggregator for 10 and 100 passes, Worker/Aggregator ignoring the job setup cost.

With these two concepts, is possible to implement Algorithm 1 more efficiently by choosing $f_{D_i}(v)$ to be a SGD-pass over the data D_i with initial model v and $g(\{v'_1, \ldots, v'_i\}) := \frac{1}{k}\sum_{i=1}^{k} v'_i$.

4 Experiments

The objectives of the experiments are (a) to show empirical convergence of the parallel SGD and (b) to study the speed differences between the MapReduce and the Worker/Aggregator implementation.

Implementation: We implemented Worker/Aggregator on top of Hadoop. This allowed us to use the distributed file system and data local scheduling facilities of this MapReduce implementation. It also facilitates code sharing between the different variants: The inner SGD loop, the data representation and all aspects unrelated to the chosen parallelization framework are shared between the implementations. The communication between Workers and the Aggregator is setup using ZooKeeper, a high-performance coordination service. The data exchange is then handled through direct TCP/IP connections. To facilitate reproducibility, all experiments where conducted on a private cluster consisting of 8 commodity machines, each equipped with 4 GB of main memory.

Data: We created a model for the data as follows. The label was selected uniformly at random, positive or negative. For each label, there was a multinomial distribution over 1 million tokens. The probability of each feature was drawn at random from $[0.9, 1] \times 10^{-6}$, with the remainder being a probability on a "ϵ token" (which generates nothing). We drew 1000 tokens for each example.[3] We generated 2 million examples. The dataset was partitioned into 8 parts, one per machine. In all experiments, a linear model of 8 MB size was trained.[4]

4.1 Computational Performance

To determine the runtime per pass over the data, we ran each of the implementations for a varying number of passes and measured the average time it took to obtain a model start to finish, including job setup costs. See Figure 1 for a graph of the results. The relatively small speedup for the MapReduce implementation on 8 machines versus the sequential version on only one machine is surprising, but explainable: The sequential version doesn't suffer from the multiple job setup costs associated with MapReduce. Those setup costs, and the data loading, is amortized across iterations in the Worker/Aggregator implementation. Also, the communication between iterations is direct, and not through files as in MapReduce. These contribute to the massive speedup of that implementation. The initial setup costs also has much less of an impact for longer runs, as expected. Note

[3] If ϵ is selected 50 times from the 1000 draws for a message, there are 950 tokens in the example.

[4] Note that this is not the most favorable setup for the Worker/Aggregator scheme, as the amount of training data per machine is relatively small, increasing the relative influence of the communication cost on the results.

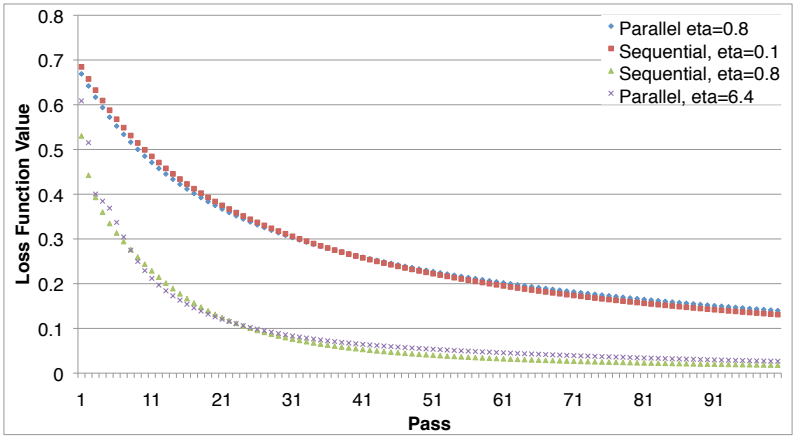

Figure 2: Training error per iteration of the sequential and parallel implementations.

that for bigger datasets, and especially for clusters with more available memory per machine, the Worker/Aggregator approach should perform even better.

4.2 Convergence

In these experiments, we ran both the sequential and the Worker/Aggregator implementation for 100 passes over the dataset in order to analyse the empirical convergence properties of the parallel algorithm. Figure 2 shows the convergence of the two algorithms for different choices of the learning rate eta. Using a eight times higher learning rate, the parallel algorithm makes as much progress per pass as the sequential one. This can be explained by the averaging step: The steps made by individual machines are all reduced by a factor of $\frac{1}{\#machines}$. Together with the huge speedups (up to 30x) demonstrated above for this setup, massive improvements in wall clock time are to be expected in applications.

5 Conclusions and Future Work

We presented a parallel framework inspired by MapReduce and MPI that enjoys both the convenience of MapReduce and the performance of message passing systems. We showed the practicality of the approach by implementing a parallel SGD inspired by [5] in both MapReduce and the proposed Worker/Aggregator framework. Our experimental results confirm convergence of the new algorithm at essentially the same rate per pass as a sequential SGD. Also, the experiments demonstrated the massive speedups possible by using Worker/Aggregator as opposed to MapReduce or even sequential SGD. We will expand on this notion in our future work by investigating additional machine learning algorithms and real-world datasets, starting with batch gradient descent.

References

[1] C.T. Chu, S.K. Kim, Y. A. Lin, Y. Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hofmann, editors, *Advances in Neural Information Processing Systems 19*, 2007.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[3] J. Langford, A.J. Smola, and M. Zinkevich. Slow learners are fast. arXiv:0911.0491, 2009.

[4] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1231–1239. 2009.

[5] Martin Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23 (to appear)*, 2010.